The Heat Equation: A study of Numerical Methods

-:Anubhav Singh:-

1. Introduction

The report discusses the numerical methods used to solve the heat equation for a 3D Heat Sink. An unstructured grid made up of 3D tetrahedron elements was used as the model for heat sink. The heat equation is a type of partial differential equation(PDE) which can solved using numerical methods. To solve the equation, the Galerkin method of weighted residuals was applied on the finite elements in the spatial domain and the Implicit Euler method was used to perform the time marching. The objective was to design a system to study temperature variation(over-time and spatially) in the 3D-models, given particular environmental settings (boundary conditions and heat sink material etc...). In this particular case, a constant heat flux(from CPU or other heat source) from the bottom and ambient air surrounding the heat sink on all other sides was assumed. The heat sink was assumed to be of copper which provided values for thermal conductivity, mass density and specific heat capacity.

The report also discusses the relevant aspects of parallelization. A 3D- model can contain millions of points, and the processing is bound to take more CPU time and memory resources. In serialized execution, this may amount for hours of calculation. The OpenCL kernel methods approach to parallelize certain portions of calculation steps in Conjugate Gradient method(used for solve the system of linear equation at each time-step) has been discussed in the report. Furthermore, the assembly of matrix **K**, **M** and **s** using the Galerkin method of weighted residuals can be parallelized by splitting the 3D model into smaller portions and executing them at multiple processes. The interprocess boundary conditions for the points shared between two processes can be defined and necessary data shared between processes.

A. The Heat Equation:

The heat equation is of the form of a PDE, which derives from the principle of conservation energy.

$$\rho C \frac{\partial T}{\partial t} = k \nabla^2 T \qquad (.1)$$

In the programmatic contruct, we used ρ = 8954 $kg\ m^{-3}$, C = 380 $J\ Kg^{-1}\ K^{-1}$and k = 386 $m^{-1}\ K^{-1}$, which are specific to copper. The neumann boundary conditions were assumed at the bottom with

a constant heat flux of $Q_{base} = k\nabla T = 10000\ W\,m^{-2}$, and ambient air on the other faces $Q_{air} = k\nabla T = h(T - T_{air})$ with $T_{air} = 300\ K$, representing robin(mixed) boundary conditions.

B. Galerkin method:

The finite element method used to solve the equation was the Galerkin method of weighted residuals. In the general method of weighted residuals, if $r(x)$ is residual function, the integral of residual multiplied by a weighting function over the entire of domain should be set to 0.

$$\int_{\Omega} W(x)r(x)=0 \qquad (.2)$$

Here, $W(x)$ is the weighting function and $r(x)$ is the residual function.

The residual function is obtained by putting plugging in the trial solution in the PDE we wish to solve. We assume the trial solution to be $\phi(x)=\sum_{n=0}^{N} a_n p_n(x)$.

In the Galerkin method, the weighting function , $W(x)$, is set to trail function $p_n(x)$.

$$\int_{\Omega} p_n(x)r(x)=0 \qquad (.3)$$

C. Application on the Heat Equation:

The assumed solution inclusive of the temporal domain is represented as:

$\phi(x,t)=\sum_{n=1}^{N} \eta_n(x)\phi_n(t)$ where, n_n are share shape functions. (.4)

Note:- We shall substitute ϕ with T for the remaining portion of the report as we are solving for temperature(T) in the heat equation.

To apply the concept of weighted residuals on the Heat Equation PDE, we would set the Weighting function as the shape function, and derive the residual function by plugging in the value from Eq(.4) in the heat equation.

$$\int_{\Omega} W_n\left(\rho C\frac{\partial T}{\partial t}-k\nabla^2 T\right)d\Omega=0$$

The trail solution $T(x)=\sum_{n=0}^{N} a_n p_n(x)$ is assumed to be vary linearly, that is (in 3D) $T(x,y,z)=a_0+a_1x+a_2y+a_3z$. But this implies that the second derivative in any of the dimensions would be non-existent $\nabla^2 T=0$. To resolve this issue, the equation is transformed into what is called “the weak form”.

Derivation of the weak form of the Heat Equation:

The weighted residual form of the equation is:

$$\int_{\Omega} W_n(\rho C\frac{\partial T}{\partial t}-k\nabla^2 T)\,d\Omega=0$$
$$\int_{\Omega} W_n\rho C\frac{\partial T}{\partial t}\,d\Omega-\int_{\Omega} W_n k\nabla^2 T\,d\Omega=0 \qquad (.5)$$

The product rule of differentiation is $\nabla(fg)=\nabla(f)g+\nabla(g)f$. Now, applying this to the second expression , $k\nabla.(W_n\nabla T)$, we get:

$$k\nabla.(W_n\nabla T)=k\nabla W_n.(\nabla T)+W_n k\nabla^2 T$$

Rearranging the terms:

$$W_n k\nabla^2 T=k\nabla.(W_n\nabla T)-k\nabla W_n.(\nabla T)$$

Substituting in Eq(.5)

$$\int_{\Omega} W_n\rho C\frac{\partial T}{\partial t}\,d\Omega-\int_{\Omega} k\nabla.(W_n\nabla T)\,d\Omega+\int_{\Omega} k\nabla W_n.(\nabla T)\,d\Omega=0 \qquad (.6)$$

From divergence theorem,

$$\int_{\Omega}\nabla.(f)\,d\Omega=\int_{\Gamma} f.d\Gamma$$

Using this in the 2nd expression of Eq(.6) we get:

$$\int_{\Omega} k\nabla.(W_n\nabla T)\,d\Omega=\int_{\Gamma} k W_n\nabla T.d\Gamma$$

Substituing in the Eq(.6) :

$$\int_{\Omega} W_n\rho C\frac{\partial T}{\partial t}\,d\Omega-\int_{\Gamma} k W_n\nabla T.d\Gamma+\int_{\Omega} k\nabla W_n.(\nabla T)\,d\Omega=0$$

Rearranging the terms:

$$\int_{\Omega} W_n\rho C\frac{\partial T}{\partial t}\,d\Omega+\int_{\Omega} k\nabla W_n.\nabla T\,d\Omega=\int_{\Gamma} k W_n\nabla T.d\Gamma$$

The term on the right hand side is the intergral over the boundary. Now, there are two boundary conditions for the Heat Equation, the robin boundary condition for the portion of heat sink in contact with the air, and neumann boundary condition on the base of the heat sink, with $Q_{base} = k\nabla T = 10000\ W\,m^{-2}$ and $Q_{air} = k\nabla T = h(T - T_{air})$ with $T_{air} = 300\ K$. Considering the boundary conditions we get the final equation of the **weak form of the Heat Equation.**

$$\int_{\Omega} W_n \rho C \frac{\partial T}{\partial t} d\Omega + \int_{\Omega} k \nabla W_n . \nabla T \, d\Omega = \int_{\Gamma_{base}} W_n Q_{base} . d\Gamma - \int_{\Gamma_{air}} W_n h(T - T_{air}) . d\Gamma \qquad (.7)$$

2. Methods

The weak form of the equation removes the second derivatives and we can apply Galerkin method of weighted residuals, considering linear variation in T within an element in spatial domain. The next step is to derive the derivatives of the assumed form of solution.

A. The Shape functions in Galerkin method

From Eq(.4), $T(x,t) = \sum_{n=1}^{N} \eta_n(x) T_n(t)$. The 3D model under consideration is that of a unstructure grid of tetrahedrons. Each element is a tetrahedron and faces are triangular. A tetrahedron elements has four points, hence the number of shape functions per element would be four.

The first derivative of Eq(.4) is :

$$\frac{\partial T}{\partial x_i} = \sum_{n=1}^{N} \frac{\partial (\eta_n T_n)}{\partial x_i}$$

Upon applying product rule, since T_n is only dependent on time and independent of spatial domain :

$$\frac{\partial T}{\partial x_i} = \sum_{n=1}^{N} \frac{\partial n_n}{\partial x_i} T_n$$

Substituting in Weak Form:

Using the above formulation and substituting the trial solution and shape functions in Eq(.7), for only one element.

$$\int_{\Omega_e} \rho C \, \eta_p \eta_q \frac{\partial T_q}{\partial t} d\Omega_e + \int_{\Omega_e} k \nabla \eta_p . \nabla \eta_q T_q \, d\Omega_e = \int_{\Gamma_{base}} Q_{base} \eta_p . d\Gamma - \int_{\Gamma_{air}} h \eta_p \eta_q T_q . d\Gamma + \int_{\Gamma_{air}} \eta_p h T_{air} . d\Gamma$$

Now, the 2nd term in the right hand side is a function of T and would be moved to left hand side:

$$\int_{\Omega_e} \rho C \, \eta_p \eta_q \frac{\partial T_n}{\partial t} d\Omega_e + \int_{\Omega_e} k \nabla \eta_p . \nabla \eta_q T_n \, d\Omega_e + \int_{\Gamma_{air}} h \eta_p \eta_q T_n . d\Gamma = \int_{\Gamma_{base}} Q_{base} \eta_p . d\Gamma + \int_{\Gamma_{air}} \eta_p h T_{air} . d\Gamma \qquad (.8)$$

The global expression for the system of equations would be represented as below, which is basically summing it over all the elements:

$$\sum_{e=1}^{N_e}\left(\int_{\Omega_e}\rho C\,\eta_p\,\eta_q\frac{\partial T_q}{\partial t}d\Omega_e+\int_{\Omega_e}k\nabla\eta_p.\nabla T_q\,d\Omega_e+\int_{\Gamma_{air}}h\,\eta_p\,\eta_q T_q.d\Gamma\right)=\sum_{e=1}^{N_e}\left(\int_{\Gamma_{base}}Q_{base}\,\eta_p.d\Gamma+\int_{\Gamma_{air}}\eta_p hT_{air}.d\Gamma\right)$$

B. System of differential equations

The previous equation can be written in the form of system of differential equations $M\dot{T}=KT+s$.

$$M=\sum_{e=1}^{N_e}\left(\int_{\Omega_e}\rho C\,\eta_p\,\eta_q\,d\Omega_e\right) \tag{.9}$$

$$K=-\sum_{e=1}^{N_e}\left(\int_{\Omega_e}k\nabla\eta_p.\nabla\eta_q\,d\Omega_e+\int_{\Gamma_{air}}h\,\eta_p\,\eta_q.d\Gamma\right) \tag{.10}$$

$$s=\sum_{e=1}^{N_e}\left(\int_{\Gamma_{base}}Q_{base}\,\eta_p.d\Gamma+\int_{\Gamma_{air}}\eta_p hT_{air}.d\Gamma\right) \tag{.11}$$

The next step is to calculate the values of each of the expressions above for which the dervations of derivatives and intergrals of shape functions are required. To derive the expression, lets consider the trail function again.

The trial function for linear tetrahedron element can be written as:

$$T(x,y,z)=a_0+a_1x+a_2y+a_3z$$

Since there are four points in a tetrahedron, we can get a set of linear equations in four variables a_n.

$$\begin{pmatrix}T_1\\T_2\\T_3\\T_4\end{pmatrix}=\begin{pmatrix}1&x_1&y_1&z_1\\1&x_2&y_2&z_2\\1&x_3&y_3&z_3\\1&x_4&y_4&z_4\end{pmatrix}\begin{pmatrix}a_0\\a_1\\a_2\\a_3\end{pmatrix}$$ This can be solved using crammer's rule for values of a_n.

Now, we also know that $T(x,t)=\sum_{n=1}^{N}\eta_n(x)T_n(t)$. Here the expression T_n is given by the values of T_1, T_2, T_3, T_4. And the expression for η_n can be derived from coefficients of T_1, T_2, T_3, T_4 when the values of a_0, a_1, a_2, a_3 are substituted in the trial solution $T(x,y,z)=a_0+a_1x+a_2y+a_3z$. The general expression for which would be:

$$\eta_n = \begin{pmatrix} 1 & x & y & z \\ 1 & x_{n+1} & y_{n+1} & z_{n+1} \\ 1 & x_{n+2} & y_{n+2} & z_{n+2} \\ 1 & x_{n+3} & y_{n+3} & z_{n+3} \end{pmatrix}$$ where n is modulo 4.

The grad of η_n can be defined as a vector.

$$\nabla \eta_n = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & x_{n+1} & y_{n+1} & z_{n+1} \\ 1 & x_{n+2} & y_{n+2} & z_{n+2} \\ 1 & x_{n+3} & y_{n+3} & z_{n+3} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & x_{n+1} & y_{n+1} & z_{n+1} \\ 1 & x_{n+2} & y_{n+2} & z_{n+2} \\ 1 & x_{n+3} & y_{n+3} & z_{n+3} \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & x_{n+1} & y_{n+1} & z_{n+1} \\ 1 & x_{n+2} & y_{n+2} & z_{n+2} \\ 1 & x_{n+3} & y_{n+3} & z_{n+3} \end{pmatrix}$$

$$\nabla \eta_n = -\begin{pmatrix} 1 & y_{n+1} & z_{n+1} \\ 1 & y_{n+2} & z_{n+2} \\ 1 & y_{n+3} & z_{n+3} \end{pmatrix}, \begin{pmatrix} 1 & x_{n+1} & z_{n+1} \\ 1 & x_{n+2} & z_{n+2} \\ 1 & x_{n+3} & z_{n+3} \end{pmatrix}, -\begin{pmatrix} 1 & x_{n+1} & y_{n+1} \\ 1 & x_{n+2} & y_{n+2} \\ 1 & x_{n+3} & y_{n+3} \end{pmatrix}$$

$$\nabla \eta_n = \begin{pmatrix} z(2)*(y(3)-y(4))+z(3)*(y(4)-y(2))+z(4)*(y(2)-y(3)) \\ z(1)*(y(4)-y(3))+z(3)*(y(1)-y(4))+z(4)*(y(3)-y(1)) \\ z(1)*(y(2)-y(4))+z(2)*(y(4)-y(1))+z(4)*(y(1)-y(2)) \\ z(1)*(y(3)-y(2))+z(2)*(y(1)-y(3))+z(3)*(y(2)-y(1));; \\ z(2)*(x(4)-x(3))+z(3)*(x(2)-x(4))+z(4)*(x(3)-x(2)) \\ z(1)*(x(3)-x(4))+z(3)*(x(4)-x(1))+z(4)*(x(1)-x(3)) \\ z(1)*(x(4)-x(2))+z(2)*(x(1)-x(4))+z(4)*(x(2)-x(1)) \\ z(1)*(x(2)-x(3))+z(2)*(x(3)-x(1))+z(3)*(x(1)-x(2));; \\ y(2)*(x(3)-x(4))+y(3)*(x(4)-x(2))+y(4)*(x(2)-x(3)) \\ y(1)*(x(4)-x(3))+y(3)*(x(1)-x(4))+y(4)*(x(3)-x(1)) \\ y(1)*(x(2)-x(4))+y(2)*(x(4)-x(1))+y(4)*(x(1)-x(2)) \\ y(1)*(x(3)-x(2))+y(2)*(x(1)-x(3))+y(3)*(x(2)-x(1)) \end{pmatrix} \quad (.12)$$

Now, **the first expression in M** can be expanded using intergration formula defined for linear tetrahedron elements,

$$\int_{\Omega_e} \eta_p^a \eta_q^b \eta_r^c \eta_s^d d\Omega = \frac{a!b!c!d!6\Omega_e}{(a+b+c+d+3)!} \quad (.13)$$

$$\int_{\Gamma_e} \eta_p^a \eta_q^b \eta_r^c d\Gamma = \frac{a!b!c!2\Gamma_e}{(a+b+c+2)!} \quad (.14)$$

where Γ_e represents area of tetrahedron, and Ω_e the volume.

$$\int_{\Omega_e} \rho C \eta_p^1 \eta_q^1 d\Omega = \rho C \frac{1!1!0!0!6\Omega_e}{(1+1+0+0+3)!} = \rho C \frac{\Omega_e}{20}$$, when, p ≠ q

$$\int_{\Omega_e} \rho C \eta_p^1 \eta_p^1 d\Omega = \rho C \frac{2!0!0!0!6\Omega_e}{(2+0+0+0+3)!} = \rho C \frac{2\Omega_e}{20}$$, when, p = q

Since the values of p and q can vary from 1 to 4, the above can be represented in the form of matrix as:

$$M_e = \rho C \frac{\Omega_e}{20} \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix}$$

Now, considering the expression of **K** in Eq(.10), **the first expression** $\int_{\Omega_e} k \nabla \eta_p . \nabla \eta_q d\Omega_e$ can be calculated by doing the dot product of the $\nabla \eta_n$ values from Eq(.12).

Whereas the **2nd expression** in **K** can be represented as :

$$M_{e-air} = h \frac{\Gamma_e}{12} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

This was derived using the Eq(.14)

$$\int_{\Gamma_e} \eta_p^1 \eta_q^1 d\Gamma = \frac{1!1!0!2\Gamma_e}{(1+1+0+2)!} = \frac{\Gamma_e}{12}$$, when p ≠ q

$$\int_{\Gamma_e} \eta_p^1 \eta_q^1 d\Gamma = \frac{2!0!0!2\Gamma_e}{(2+0+0+2)!} = \frac{2\Gamma_e}{12}$$, when p = q

Lastly, for the expression $s = \sum_{e=1}^{N_e} \left(\int_{\Gamma_{base}} Q_{base} \eta_p . d\Gamma + \int_{\Gamma_{air}} \eta_p h T_{air} . d\Gamma \right)$, using the intergration formula defined for linear tetrahedron elements again Eq(.14)

$$\int_{\Gamma_{base}} Q_{base} \eta_p . d\Gamma + \int_{\Gamma_{air}} \eta_p h T_{air} . d\Gamma = Q_{base} \int_{\Gamma_{base}} \eta_p^1 . d\Gamma + \int_{\Gamma_{air}} \eta_p h T_{air} . d\Gamma$$

The expression $\int_{\Gamma_e} \eta_p^1 . d\Gamma = \frac{1!0!0!2\Gamma_e}{(1+0+0+2)!} = \frac{\Gamma_e}{3}$

With this in place we get the required values for our system of equation $M\dot{T} = KT + s$. The next step involved solved the equation using Implicit Euler Method.

C. Implicit Euler Method

The implicit Euler method follows from the Explicit Eulers method with the only difference that $f(\phi^{l+1}, t^{l+1})$ is calculated at the time-step $l+1$ instead of l. The basis of Euler's method lies in the Taylor series expansion.

$$\phi(t+\Delta t) = \phi(t) + \Delta t \frac{\partial \phi(t)}{\partial t} + \frac{\Delta t^2}{2!} \frac{\partial^2 \phi(t)}{\partial t^2} + \frac{\Delta t^3}{3!} \frac{\partial^3 \phi(t)}{\partial t^3} + \ldots$$

Neglecting the 2nd order and higher terms:

$$\phi(t+\Delta t)=\phi(t)+\Delta t\frac{\partial\phi(t)}{\partial t}$$

$$\phi(t+\Delta t)=\phi(t)+\Delta t f(\phi(t),t)$$

In the implicit euler's method the expression changes to

$\phi(t+\Delta t)=\phi(t)+\Delta t f(\phi(t+\Delta t),t+\Delta t)$ which can also be shown as:

$$\phi^{l+1}=\phi^{l}+\Delta t f(\phi^{l+1},t^{l+1})$$

For a system of partial differential equation, we have:

$$M\frac{\phi^{l+1}-\phi^{l}}{\Delta t}=K\,\phi^{l+1}+s$$

This can be converted into a system of linear equations $A\,\phi^{l+1}=b$, where $A=M-\Delta t\,K$ and $b=M\,\phi^{l}+\Delta s$.

Now at each time step the value of A would not change since M and K are fixed. So, the only expression which needs to be calculated is *b.* The system of equation $A\,\phi^{l+1}=b$ was solved directly in matlab and in C++ code the conjugate gradient method was used.

D. Data Structures

The sheer number of elements in a 3d grid structure requires efficient storage of data. What was required was data structures which can efficiently store data, without causing lot of processing overhead in data retreival and retreival.

A mixture of 1D and 2D arrays and sparse matrix were used in the program for efficient storage of data as discussed below.

1. Sparse matrix data-structure for M and K matrix

The sparse matrix was used to store the matrices **M** and **K** in the system of differential equation. These matrices can be very sparse depending upon the boundary condition and equation at hand. Compressed row storage technique was used to design the sparse matrix. In this, the non-zeros values are stored in one array. A second array captures the index where the data for each row in original matrix is stored. A third array, which is the same size as the 1st array, captures the actual column number of each value in 1st array.

All the arrays in Sparse Matrix can increase in size at runtime. However, this may cause performance degradation as the sparse matrix becomes more and more un-sparse, since the arrays need to be expanded in size everytime the number of non-zero values increase beyond the allocated size. Further more, reading and writing are essentially reading 3 arrays instead of 1, to read a value in actual array. So, the reads and writes become slower as well.

In conclusion, the real potential of this data-structure is realized in very big but very sparse matrices.

2. Storage of grid data (points, faces and elements)

The points in the grid vtk files had corresponding values for each of the coordinate axis *x, y, z.* Similary an Element captured the 4 points of tetrahedron and the Faces captured the 3 points of a face. The data-structure used to store this information were 1D contiguous array, for faster read access. For example, all the information of points was stored in a 1D array of size 3*N_p, where in the pointer P[i] was set to every third element in the 1D array. So that the object could be retreived by calls similar to 2D arrays. The continguous allocation of memory allows for faster read-write during execution.

E. OpenCL parallization

The OpenCL library was used to develop parallel code for two problems, 'daxpy' $Y = aX + Y$ and 'inner product' $Y = X^T X$. This could be used in the conjugate gradient method where there are few calculations which are performed at every time-step to solve the system of PDE, $A\,\phi^{l+1} = b$, which can be parallelized with above code.

The following equation represent the values which are calculated at each time-step in conjugate gradient method.

$$\phi^{k+1} = \phi^k + \alpha d^k \quad , \quad r^{k+1} = r^k - \alpha d^k \quad , \quad \beta = \frac{\left(r^{k+1}\right)^T r^{k+1}}{\left(r^k\right)^T r^k} \quad \text{and} \quad d^{k+1} = r^{k+1} + \beta d^k$$

The 1st, 2nd and 4th equation can be parallelized using the 'daxpy' method, where as both the numerator and denominator of 3rd equation can be calculated used the inner-product method. Each of the parameters here are 1D-vector stored in an array of size *<number of points>.* The parallelization is beneficial here since these can be very large vectors with hundred thousands of points. This is specially true for 'daxpy' method since the entire calculation can be parallelized in one step. However, for the inner product the product of individual items has to be summed up as well, which leads to some amount of serialization in execution of the OpenCL Code.

3. Results

A. Matlab Solution

This section contains the screenshots of matlab simulation results at different time stamps.

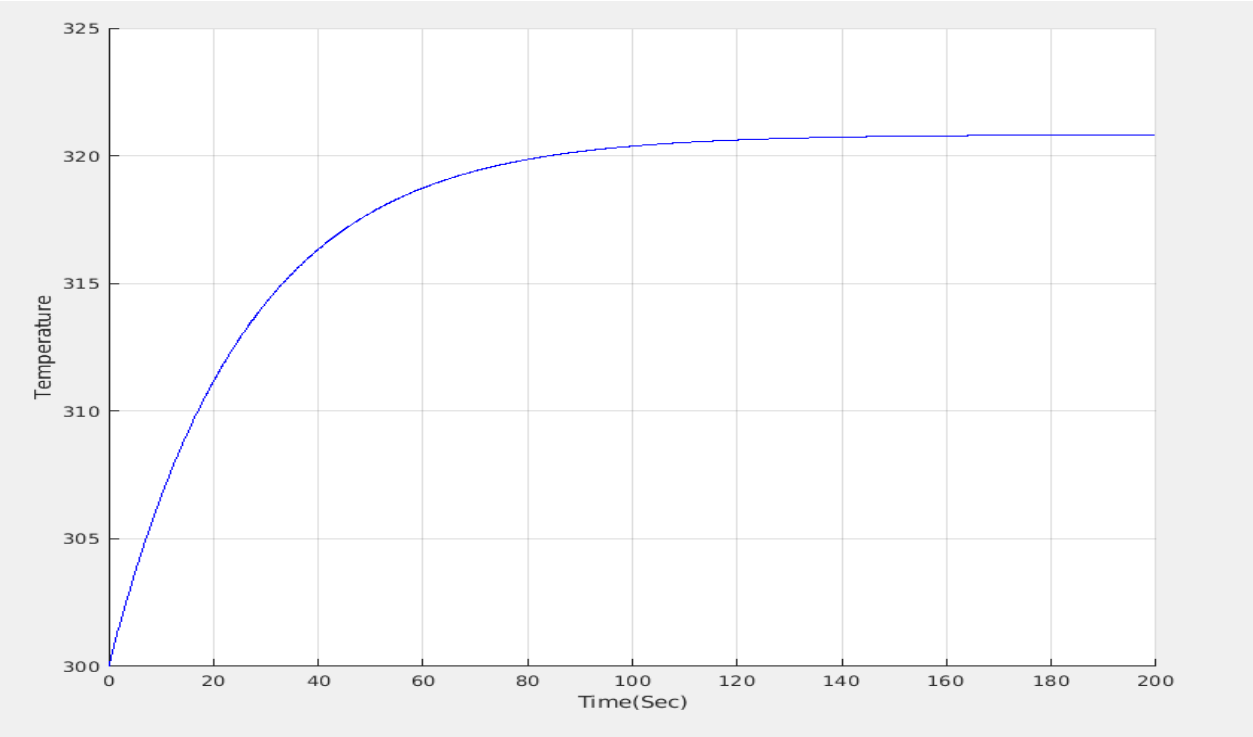

Illustration 1: Plot of Max Temperature against Time

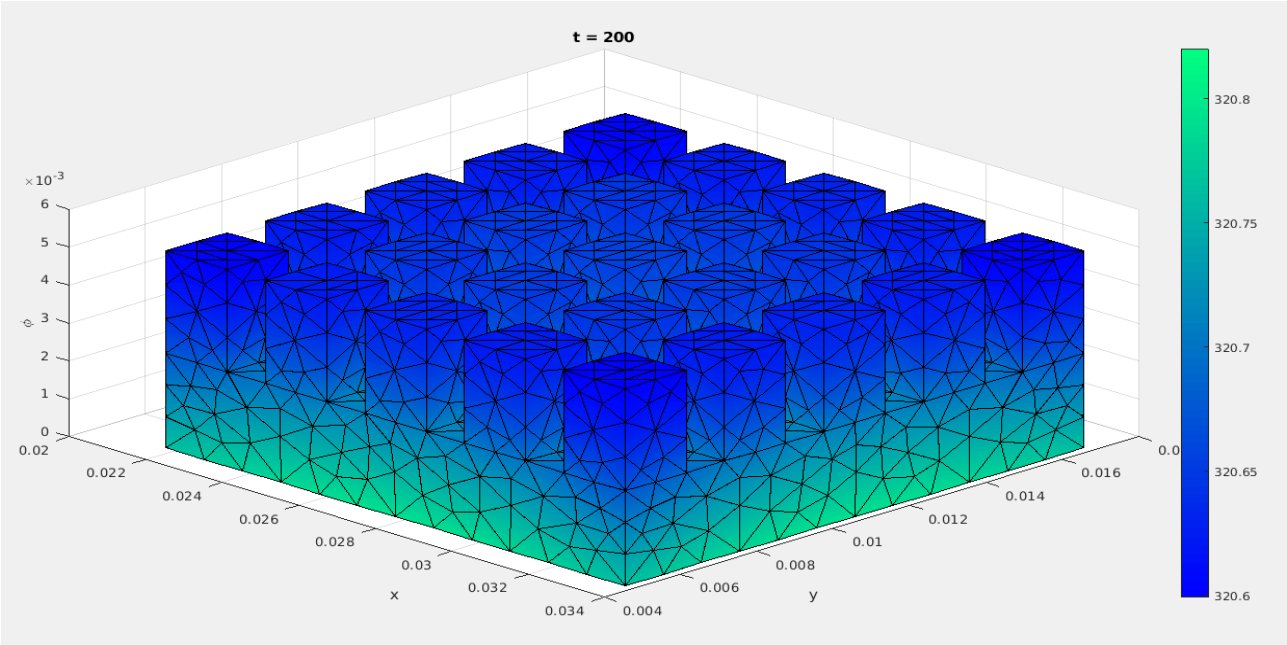

Illustration 2: The matlab simulation result at T ~ 200 seconds

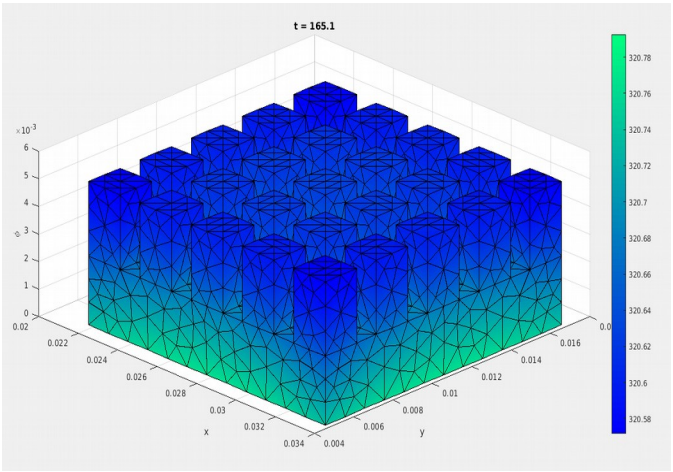

Illustration 4: The matlab simulation result at T ~ 165 seconds

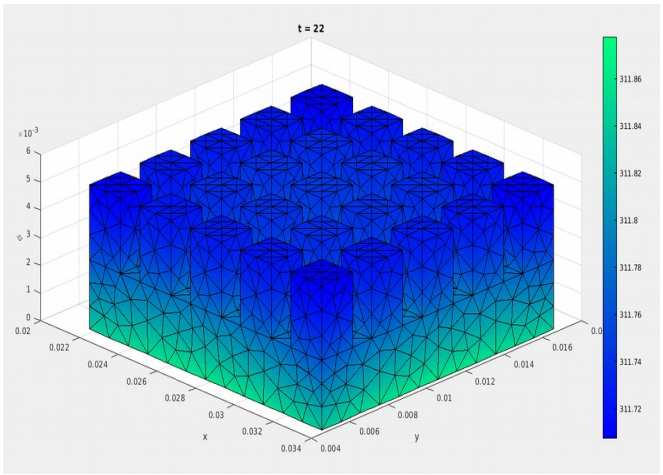

Illustration 3: The matlab simulation result at T ~ 22 seconds

B. C++ code VTK output

The following section contains screen shots of VTK file models for **5k elements grid file**.

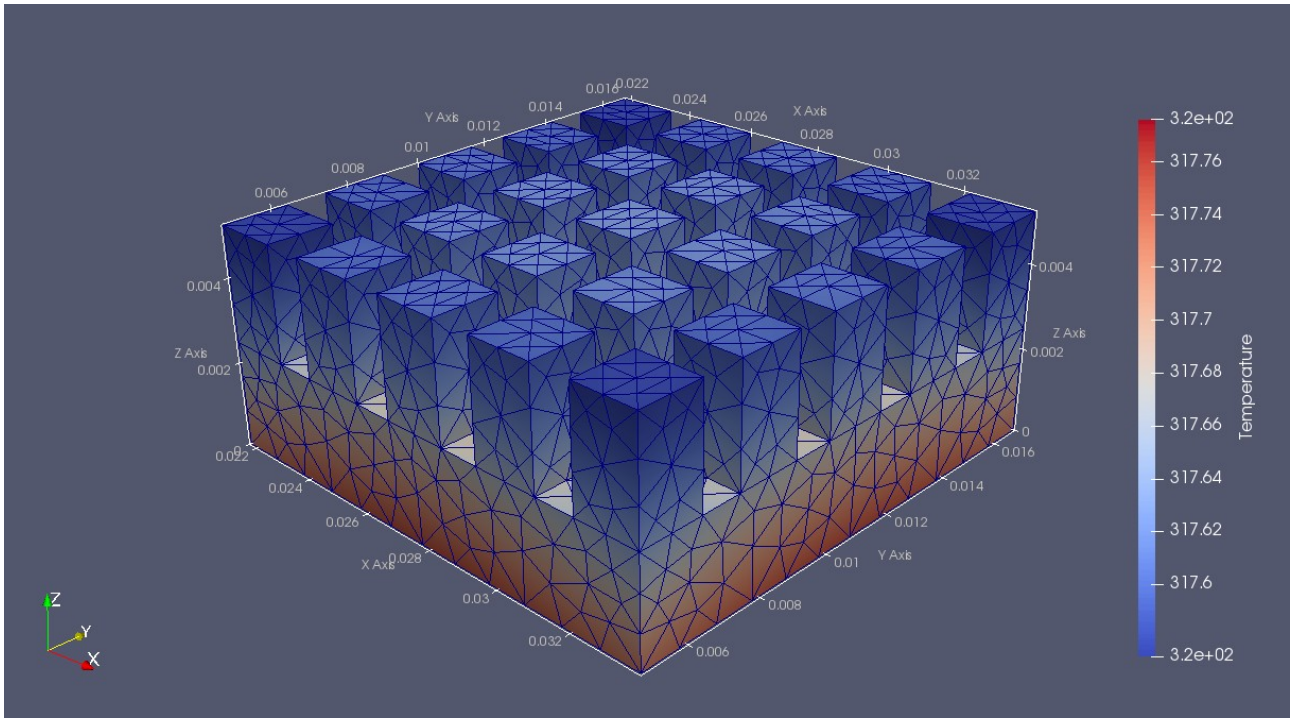

Illustration 5: C++ code output representation at T = 50 sec for 5k elements grid file

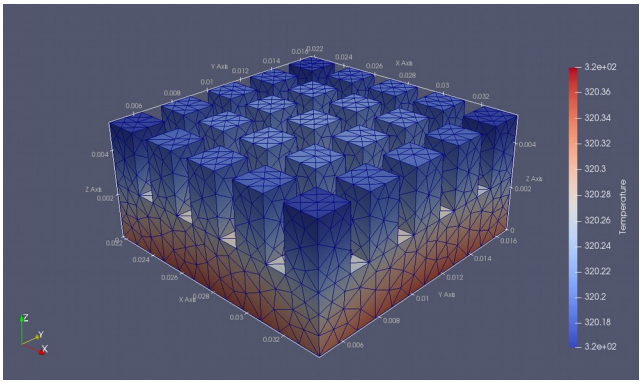

Illustration 6: C++ code output representation at T = 100 sec for 5k elements grid file

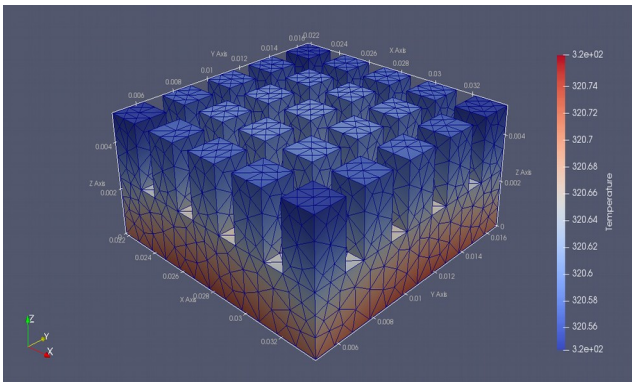

Illustration 7: C++ code output representation at T = 150 sec for 5k elements grid file

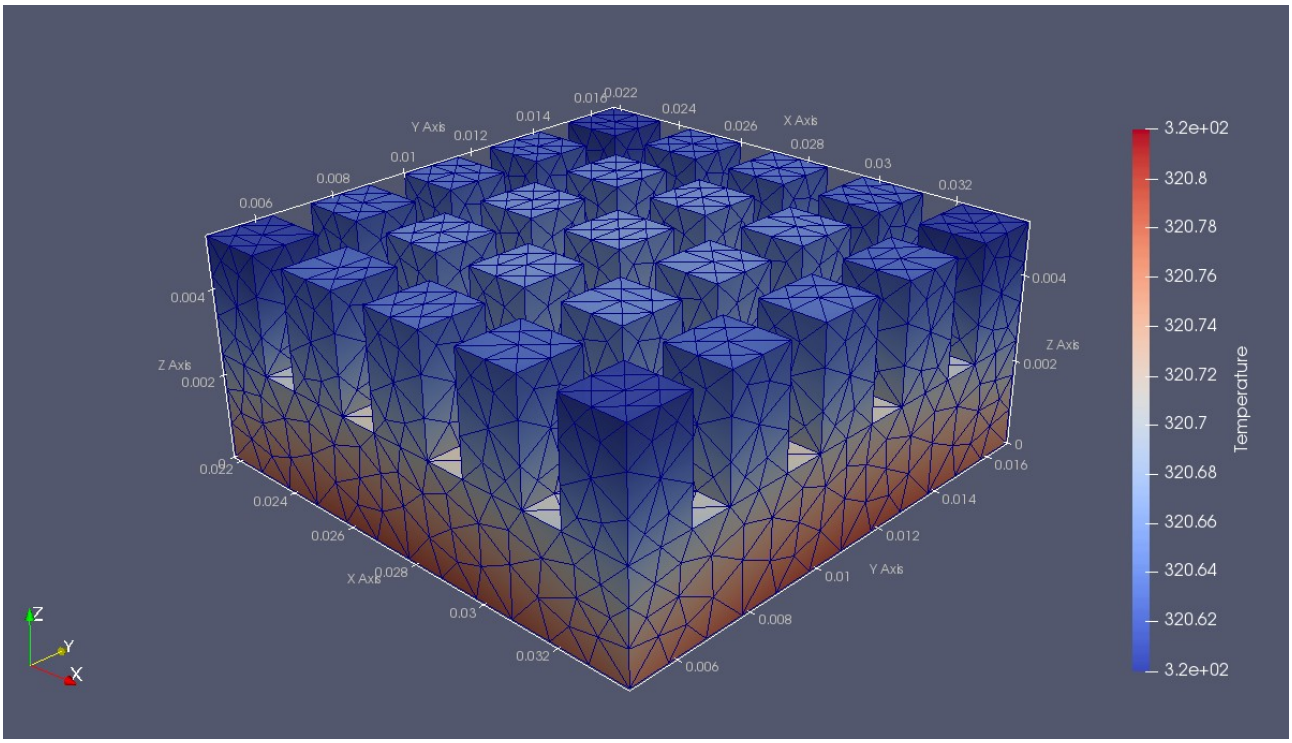

Illustration 8: C++ code output representation at T = 200 sec for 5k elements grid file

For 30k elements grid-

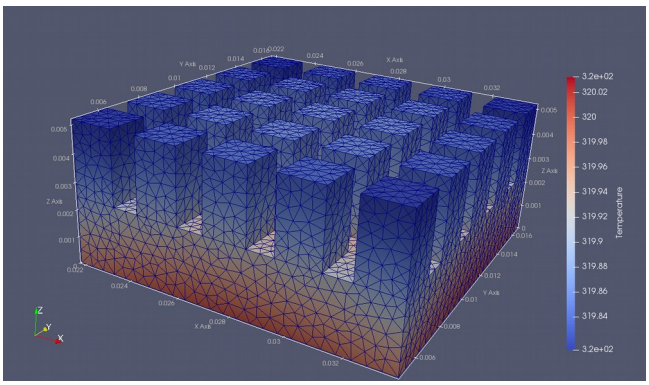

Illustration 9: C++ code output representation at T = 85 sec for 30k elements grid file

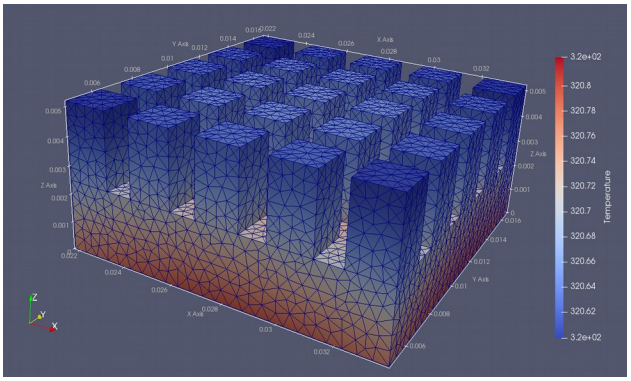

Illustration 10: C++ code output representation at T = 200 sec for 30k elements grid file

C. Performance impact analysis, varying grid-size and Δt.

A . Plot of Number of time steps vs Total execution time for different sized grids

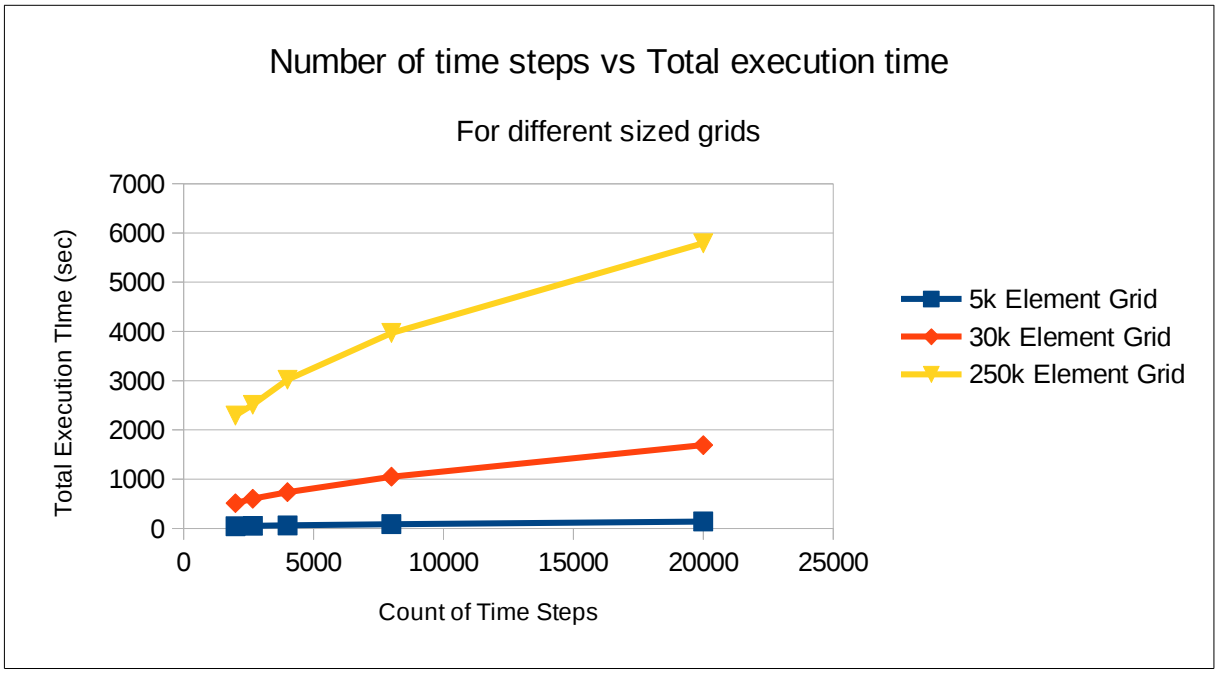

Illustration 11: Plot of Number of time steps vs Total execution time for different sized grids

	Time Taken in seconds		
Time Steps	**5k Element Grid**	**30k Element Grid**	**250k Element Grid**
2000	43.99	511.22	2291.82
2666	51.63	602.33	2510.76
4000	61.07	734.02	3020.34
8000	85.01	1049.1	3968.32
20000	138.191	1689.77	5792.1

Illustration 12: Number of time steps vs Total execution time for different sized grids

B .Table representing the Number of time steps vs Total execution time for different sized grids

D. Scaling results on OpenCL code

A. Results from barcoo supercomputing node

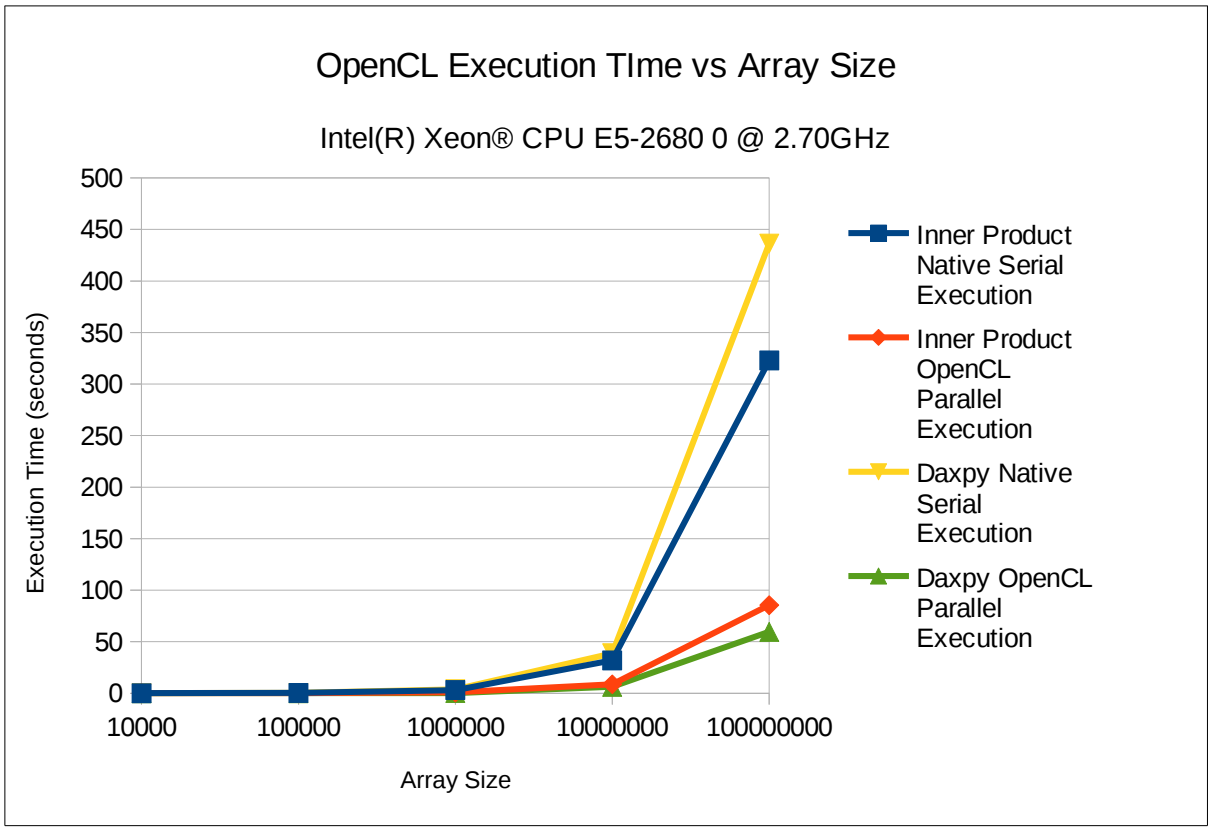

Illustration 13: Scaling results from barcoo supercomputing node

Intel(R) Xeon® CPU E5-2680 0 @ 2.70GHz	Inner Product		Daxpy	
Size of vector	Native Serial Execution	OpenCL Parallel Execution	Native Serial Execution	OpenCL Parallel Execution
10000	0.036682	0.03624	0.042677	0.028776
100000	0.339687	0.115309	0.402657	0.038863
1000000	3.06996	0.861926	3.83237	0.220207
10000000	31.9084	8.65344	38.7299	6.24261
100000000	322.839	85.4194	436.086	59.5989

Illustration 14: Scaling results from barcoo supercomputing node

B. Results on intel Graphics Node (Intel(R) Gen9 HD Graphics NEO) on Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

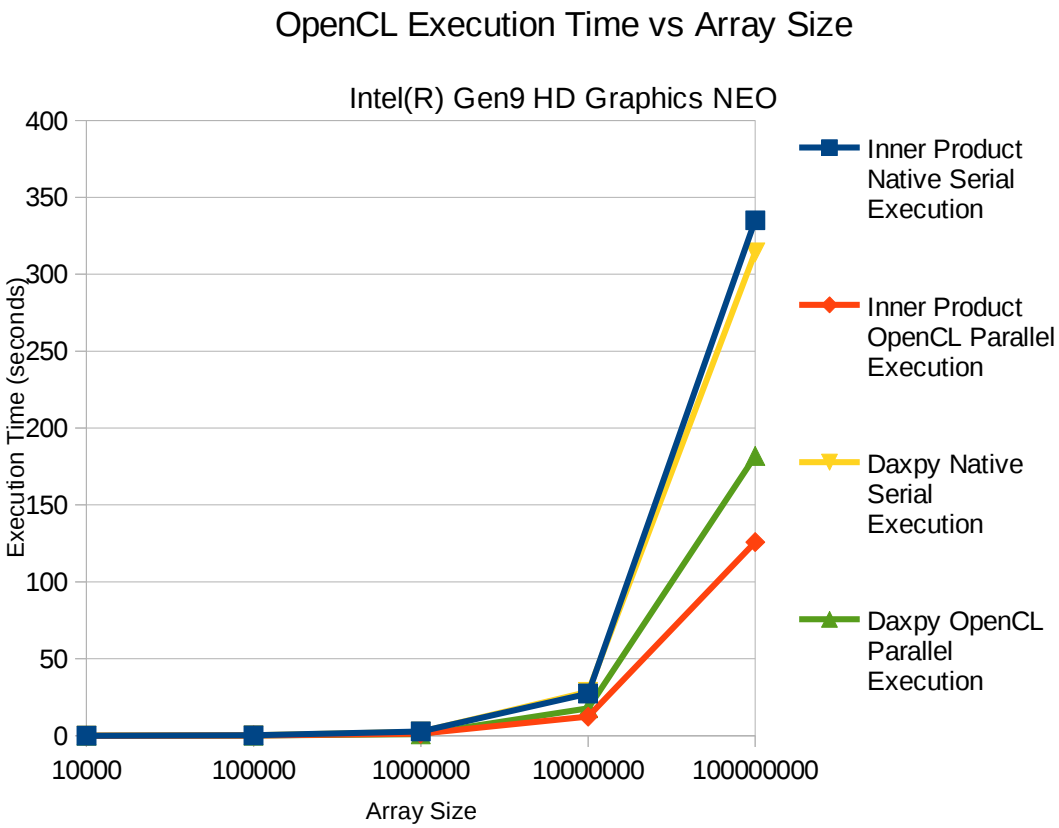

Illustration 15: Results on intel Graphics Node (Intel(R) Gen9 HD Graphics NEO) on Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

Intel(R) Gen9 HD Graphics NEO	Inner Product		Daxpy	
Size of vector	Native Serial Execution	OpenCL Parallel Execution	Native Serial Execution	OpenCL Parallel Execution
10000	0.031541	0.040687	0.024258	0.027014
100000	0.286938	0.170896	0.274127	0.086032
1000000	2.76662	1.30547	2.63082	0.88955
10000000	27.3785	12.392	28.5015	17.788
100000000	335.064	125.866	314.186	181.856

Illustration 16: Results on intel Graphics Node (Intel(R) Gen9 HD Graphics NEO) on Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz

4. Discussion

In Illustration 12, it was observed that the execution time of the C++ code for same number of time-steps increased with larger sized grid. However, the inrease in execution time was not directly proportional to the increased in grid element count. For example, for increase from 5k to 30k elements, the increase in the number of grid elements was 6 times, and the corresponding increase in execution time was *12 times.* Whereas, from 5k grid to 250k grid the increase in elements was 50 times, however the increase in execution time was also *50 times.* Upon deeper inspection of the grid files it was found that the variation was because of the actual number of points in a grid structure. While the 250k grid element file had 50 times the number of elements in comparison to 5k grid, it had only 21 times more the points(~42k). In comparison the 30k grid file had 4 times more points(~8k). The number of points in 5k grid was 1.8k. Upon looking at the data-structures on which majority of the calculation was perfomed, the size of the data-structure was directly related to the number of points in the grid structures. Only the calculation in the initial step of calculation of K, M and s are dependent on the number of elements (*for loops over all elements*). Hence, the observed results are justified.

The scaling analysis was performed on OpenCL code and Illustration 14, 15, 16 and 17 show the corresponding results. The results from barcoo supercomputing node were as expected, the 'daxpy' computation took much lesser time with the OpenCL parallelization, whereas the computation of 'Inner Product' were some what longer. This is because while the 'daxpy' can parallelized for each index in the array, 'inner product' requires summing up of the contents of each index, which requires some serialized steps. However, the native serial execution for 'daxpy' was however longer than 'Inner Product' which is also expected since each step in 'daxpy' involves both multiplication and addition steps, whereas its only multplication in inner product.

In comparison, the results on the Intel Graphics Node (Intel(R) Gen9 HD Graphics NEO) in Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz were a bit off. The calculation of 'daxpy' with OpenCL parallel code took more time than calculation of 'Inner Product' , which can basically be attributed to lesser number of parallel cores available on GPU of desktop grade processor. Hence, it may be possible that most of the calculation steps ended up being serialized. As oberved from the comparison of the serialized code, 'Inner Product' takes lesser time. Hence, the 'daxpy' with OpenCL parallel code took more time on Intel GPU.